

Time Petri Net Models for a New Queueless and Uncentralized Resource Discovery System

Camille Coti Sami Evangelista Kais Klai

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030

F-93430, Villetaneuse, France

{first.last}@univ-paris13.fr

Abstract—In this report, we detail the model using Petri Nets of a new fully distributed resource reservation system. The basic idea of the considered distributed system is to let a user reserve a set of resources on a local network and to use them, without any specific, central administration component such as a front-end node. Resources can be, for instance, computing resources (cores, nodes, GPUs...) or some memory on a server. In order to verify some qualitative and quantitative properties provided by this system, we need to model it. We detail the algorithms used by this system and the Petri Net models we made of it.

I. A NEW FULLY DISTRIBUTED RESSOURCE MANAGEMENT SYSTEM

In this section, we describe how the machines are reserved for a user. Our system is made of two parts: the *launcher* (called *qurdeli*, as “qurd client”), which is executed by the user who wants to run a job on a set of computing nodes, and the *agent* (called *qurdd*, as “qurd daemon”), which is a daemon running on all the resources that exist in the system. This architecture is depicted in figure 1.

Our algorithm relies on the service discovery tools provided by the Zeroconf protocol [1]. Computing nodes declare themselves on the Zeroconf bus, like any network service. Then a user’s launcher can look on the local Zeroconf bus which nodes are available. However, this simple discovery service is not sufficient to ensure that the computing resources will not be used by several jobs at the same time.

A. The Job Launcher

The launcher’s behavior is quite straightforward (see Algorithm 1): it reserves nodes and, as soon as it has enough nodes for its job, it starts the job on the said nodes.

Algorithm 1: Job start-up general algorithm

Data: Job J to execute on N nodes

M = reserve_nodes(N);

if card(M) == N **then**
 start_job(J, M);

The reservation algorithm uses the service discovery features of Zeroconf. It listens on the Zeroconf bus and finds out which computing nodes have published themselves (as available). The information published by each node includes the host’s name and a port on which the node can be contacted. Then, for each node it has discovered, the launcher contacts

the machine on the declared port. If the node is available, the launcher receives an acknowledgement (“OK”); otherwise it receives a rejection (“KO”). In the former case, it inserts the node on its set of reserved nodes. As soon as the set of reserved nodes contains the required number of computing nodes, it starts launching the job on these nodes. In case there is not enough available resources, we can define two different behaviors: either the reservation returns the (smaller) set of resources it was able to reserve, or it waits until it gets all the necessary resources (see Algorithm 3).

Algorithm 2: Resource reservation algorithm (fail semantics)

```

reserveNodes( nbNodes) begin
  Data: machines = {}
  listenZeroconf();
  foreach machine m newly discovered do
    if card( machines ) < nbNodes then
      contactMachine( m );
      ack = receiveAck( m );
      if ack == OK then
        machines.append( m );
  if card( machines ) == nbNodes then
    return machines ;
  else
    freeMachines( machines ) ;
    return {} ;

```

B. The Computing Resources (Machines)

When it is available (*i.e.*, it can be used to run a job), a machine publishes itself on the Zeroconf bus. When it switches to the unavailable state, it unpublishes itself. However, due to the latency of Zeroconf and to a possible concurrency between the users, the machines also need to implement a local reservation system to make sure that they are used by only one job at a given moment.

Each machine executes Algorithm 4 in order to interact with clients and to execute their jobs when possible. A machine can be in one of the three following states: *available* (the machine is free, it can be reserved for a job); *reserved* (the machine is reserved for a job but the job is not running on it yet); *running*

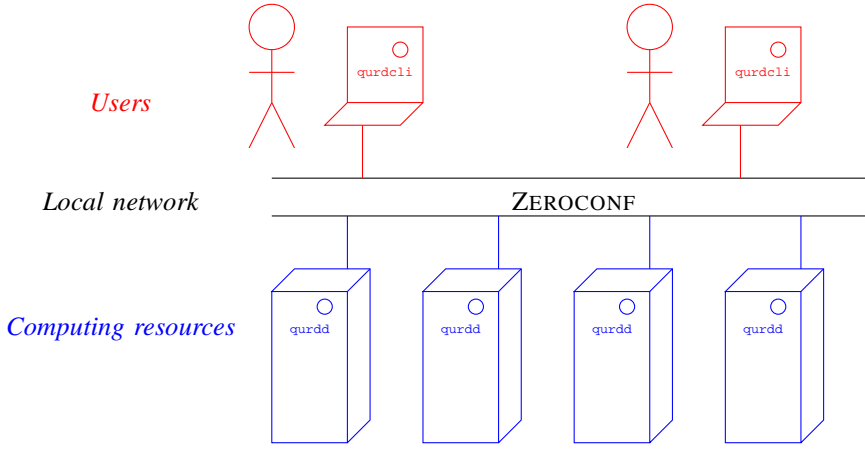


Fig. 1: Architecture of a QURD system.

Algorithm 3: Resource reservation algorithm (wait semantics)

```

reserveNodes( nbNodes) begin
  Data: machines = {}
  while card( machines ) < nbNodes do
    listenZeroconf();
    foreach machine m newly discovered do
      if card( machines ) < nbNodes then
        contactMachine( m );
        ack = receiveAck( m );
        if ack == OK then
          machines.append( m );
  return machines ;

```

(a job is running on the node).

When a machine is in the state *available*, it can be reserved by a user. The user is then considered by the machine as its *client*. Upon request reception, it sends an acknowledgement to the client, switches into *reserved* state and unpublishes itself from Zeroconf. If a machine receives a request from a client while it is not in *available* state, it sends a rejection to the client. When a machine is in the state *reserved*, it can receive a job from its client. The machine executes this job until completion. Then it switches into the state *available* and publishes itself back on Zeroconf.

It can happen that the client does not obtain the required number of machines. In this case, when at state *reserved*, a machine can be released which makes it switch to the state *available* (the machine publishes itself back on Zeroconf).

II. PETRI NETS-BASED MODEL OF THE RESSOURCES RESERVATION SYSTEM

In the following, we consider that an *application* (or a *job*) is made of several *processes* that are meant to run on a set of *resources*, also called *machines*. The user submits an application through a *client*. For the sake of readability,

we describe the model of each component of the system separately. The reader can find in the annex a full model involving one application and four resources. A coloured Petri net model is presented as well allowing to have more compact and parameterized model.

We first describe the model of a computing resource's behaviour in section II-A, and the model of a client's behavior in section II-B. We present an optional model of Zeroconf's semantics in section II-C. Then we explain how volatility is handled in section II-D and how concurrency between jobs on a given resource is modeled in our system in section II-E. The full models are given in the next section of this paper, in figure 8, 9 and 10.

A. Model of a machine: qurdd

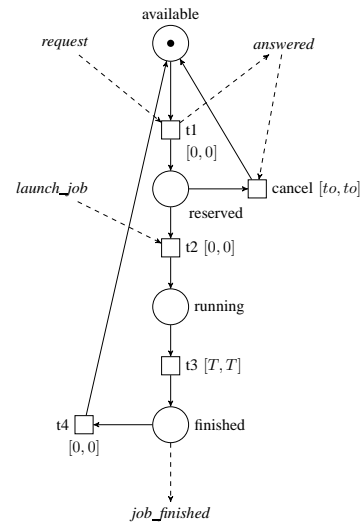


Fig. 2: Petri net model of a machine (qurdd daemon)

The Petri net model of a machine (figure 2) is guided by Algorithm 4. A machine can be reserved when it is *available*. It answers the client and switches into *reserved* mode. Starting from this state, two behaviors are possible: either the reservation is cancelled (through transition *cancel* that can be fired after some timeout) and the machine becomes available again (through a token present at place *available*), or

Algorithm 4: Algorithm executed on each resource

```

waitjob() begin
  Data: state = available
  Data: client = -1
  publishMyselfOnZeroconf();
  while True do
    received, c = rcvFromClient();
    switch received.type do
      case reservation
        if state == reserved then
          c.send( KO );
        else
          c.send( OK );
          state = reserved;
          client = c;
          unpublishFromZeroconf();
      case job
        if state == reserved OR c != client then
          refuse();
        else
          executeJob( received );
          state = running;
          publishMyselfOnZeroconf();
          state = available;
          client = -1;
      case release
        publishMyselfOnZeroconf();
        state = available;
        client = -1;
  
```

it starts executing the required application and switches to the state *running* (through the firing of transition $t2$). When this local process is done, the machine switches to state *finished*, signals to the client that its part of the job is done, and then returns back to state *available* (through transition $t4$).

B. Model of the reservation system: qurdcli

Figure 3 illustrates the reservation system of a client (Algorithm 2). For the sake of simplicity and readability, we replaced the machines by dashed boxes, each corresponds to an instance of the model represented by figure 2. When an application needs to get n resources, the *begin* place triggers a *start_job* transition whose firing produces n tokens in the *get_nodes* place. Each of these tokens corresponds to a required resource. The reservation system gets a list of the resources that are declared on the Zeroconf bus. The *get_nodes* place is an input place of all the $t1$ transitions of the machines that are present on the Zeroconf bus (see Figure 2). If the machine is actually available, a token is present in the place *available* and the transition $t1$ can be fired

leading to the reservation of the machine by this application (place *reserved* is marked). Also, the place *answered* of the reservation system being an output place of each transition $t1$, it will be marked after firing $t1$. Thus, the number of tokens present at this place represents the number of positive answers received by the machines. When the required number of resources (denoted here by n) is available (n tokens are present in place *answered*), the *launch* transition can fire producing n tokens in the place *launching_job*, corresponding to the action of actually triggering the start of the application's execution.

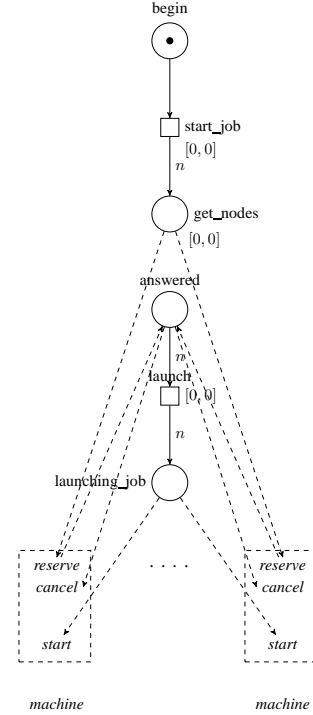


Fig. 3: Model of the reservation system

The place *launching_job* is an input place of each $t2$ transition of the machine model. If a machine has been reserved for this application, a token is present in its *reserved* place. Therefore, the $t2$ transition can be fired as soon as all the required resources answered positively and each involved machine moves to state *running* (represented by place *running*).

When the totality of the required resources is not available, the system will be in a situation where n_1 tokens are in place *answered* and n_2 tokens are in place *get_nodes* such that $n_1 > 0$, $n_2 \geq 0$ and $n_1 + n_2 = n$. In order to avoid holding these resources indefinitely, the system must free them after some timeout (which is not explicitly represented in the model but could be by associating a time interval with transition *cancel* as for Time Petri nets). When a machine stays at the state *reserved* more than the timeout, the *cancel* transition of the machine is fired, consuming a token from the *answered* place. The machine is freed (it returns into state *available*), it is removed from the list of resources that are considered by the client as having answered positively (by consuming the corresponding token from place *answered*). Finally, this timeout mechanism can be used as well when the client disconnects from the system before the application

is started (this is observed from the machine size by the fact that the application is not launched by the client after the timeout).

When the execution of a process is done, the machine exits from the state *running* and gets into state *finished*. When a token is in the resources' *finished* place, transition *t4* can be fired. The firing of *t4* produces two tokens: one of them goes into the *job_finished* place, which models the fact that this particular resource has signaled to the client the fact that its process is done, and the other one goes into the *available* place, to put the machine back into the set of machines that are available.

In section I-A we mentioned two possible semantics for the case when not enough resources are declared on the Zeroconf bus at the moment when the client issues a request. In the *fail semantics*, all the machines are released and the client does not get any resource at all. In this case, all the reserved machines fire the *cancel* transition. In the *wait semantics*, the client waits until enough machines are available, until the timeout is reached (in order to avoid deadlocks between concurrent applications). Hence, both of these semantics are modeled.

C. Modeling the Zeroconf publish/unpublish semantics

Additionally, for some reason a machine can withdraw itself from the Zeroconf bus and publish itself back. For instance, if a service suddenly takes all the memory of the server, there may not be enough memory left to host new services. Or a machine can be turned off by an administrator. An optional component of our model can describe this behavior. The model is represented by figure 4.

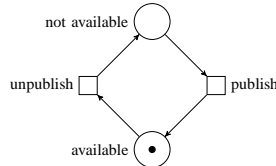


Fig. 4: Zeroconf semantics

When a machine has declared itself on the Zeroconf bus, a token is placed in its *available* place. When it is unpublished from the bus, the *unpublish* transition is fired and a token is placed in the *not_available* place. The only way it can be used is if there is a token in the *available* place. Hence, it needs to fire the *publish* transition to be used again, modeling the fact that the machine has published itself back on the Zeroconf bus.

D. Volatility

To be safe and robust, such a system must be able to handle volatility. In this section, we present how both sides (client and resources) are handled in a robust way in order to make sure that failures or disappearances have no impact on the reservation system.

1) *Client volatility*: Client volatility is due to the fact that a client can leave the network unexpectedly at any moment. The model described in section II-B already handles this fact. If a client disappears when a machine is in state *available*, it has no incidence on this machine. If the client disappears after a reservation has been issued (the machine is in state *reserved*), the timeout mechanism makes sure that the machines are freed and become available for other applications. If the client

disappears when the machine is either in state *running* or in state *finished*, the application is already running on the machines and the client can come back later to collect the results with no incidence.

2) *Resource volatility*: A resource can disappear while a process is running on this resource. Here we assume that the system is using an *eventually perfect* failure detector, such as Heartbeat [2]. We can model this failure detector by a specific place in the system: the *failure detector* place. While a machine is in state *running*, its disappearance is modeled by firing a *dead* transition, which puts a token in the place *failure detector*.

The failure detector is in charge with finding a new available machine and restarting the failed process. If a machine has a token in its place *available*, it can fire transition *restart* and set this machine directly in state *running*. It should be noted that the *restart* transitions are resource-specific, whereas the failure detector is a global component.

If a machine disappears while in state *reserved*, the timeout mechanism allows to fire transition *cancel* (see Figure 5).

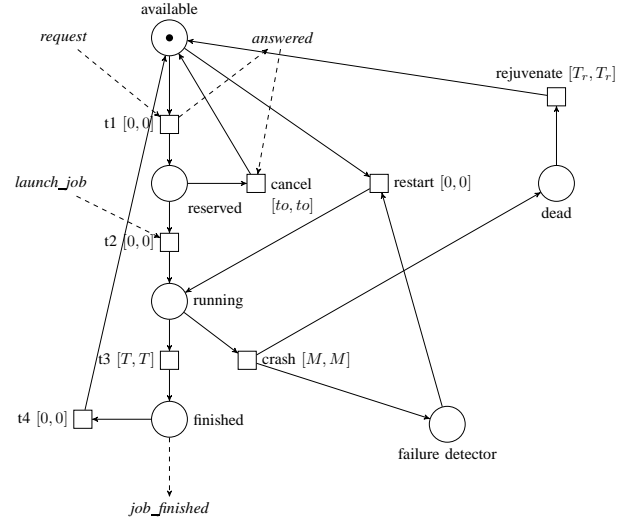


Fig. 5: Model for handling resource volatility with a failure detector

E. Concurrency between clients

When several clients are issuing resource requests, there may be a situation of concurrency between the clients. Each client browses the Zeroconf bus and gets a list of available resources. A given resource can be discovered by several clients in the same time. However, the fact that a machine can answer positively only when it is in the state *available* (before switching to state *reserved*) makes sure that it can answer positively to one client only. This is modeled by the fact that there is only one *available* place on each resource, and this place contains only one token in the initial marking.

A model where 2 clients issue concurrent resource allocation requests on the same set of resources is represented on Figure 6. Each client has its own reservation system. On the figure, we represented a client requesting n resources and another one requesting m resources.

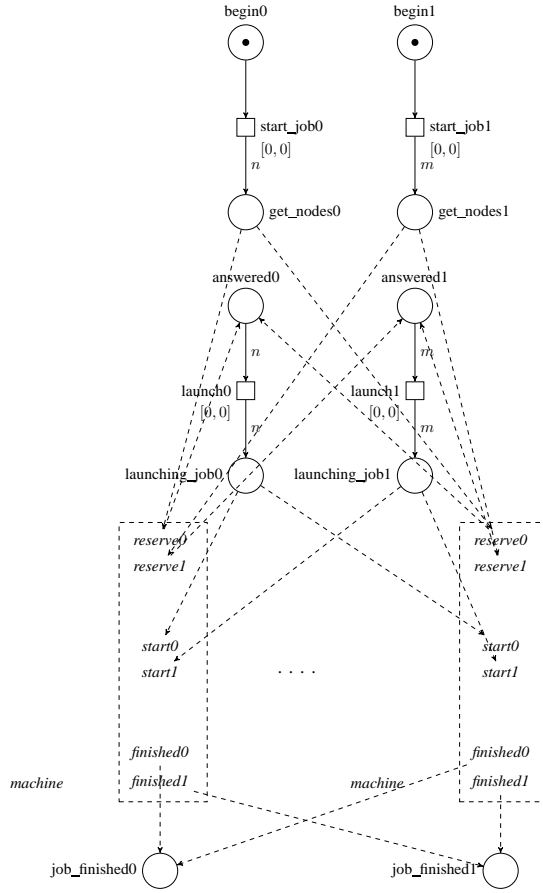


Fig. 6: Reservation system of two clients on a given set of resources

The local state of each resource depends on the application it is working for. For instance, the state *reserved* is specific to “application X”; hence, the state is actually *reserved for application X*. Therefore, the execution path of the model from the reservation to the end of the execution of the local process must be specific for each application. However, as explained earlier in this section, since the initial marking features only one token in place *available* of the machine, only one of these paths can be active at a time. The model for a machine that may answer to two applications is represented in figure 7. The left path correspond to an application called 0; the right path corresponds to an application called 1. When the resource is in state *available* and application 0 issues a request, the *t10* transition is fireable resulting for the resource to turn into state *reserved for 0* and there is no token left in place *available*. As a consequence, if application 1 issues a request, the *t11* transition cannot be fired and as long as the resource is not back into *available* mode, it will not be able to be used by application 1.

The *cancel* transition is very important here to release some resources in case of a deadlock caused by a conflict between applications in the wait semantics (see section I-A, Algorithm 3). For instance, if we have three resources and two applications asking for three and two machines respectively, then the following deadlock state can be reached: The first application gets two machines and the second one gets one machine. Thus, all the available machines are reserved but no

application is able to start. Therefore, after a certain time, if no additional resources appear on the Zeroconf bus, the machines reserved for at least one application will be freed and become available for the other one.

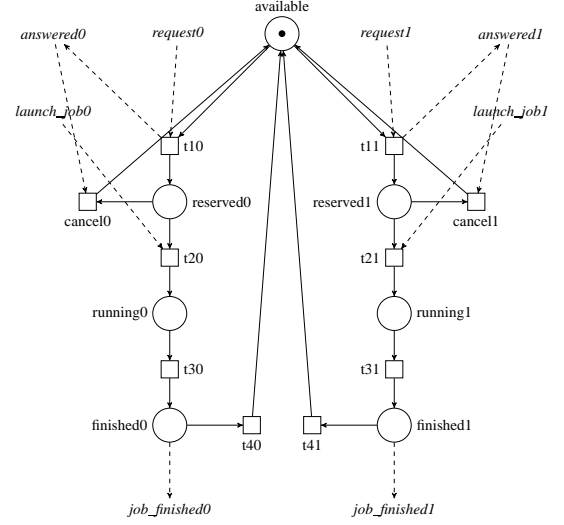


Fig. 7: Two concurrent requests on a single resource

III. COMPLETE MODELS

IV. CONCLUSION

In this report, we have described a model using Petri Nets for a decentralized resource reservation system. This model can now be used to verify properties regarding the behavior of this system. For instance, one can verify under which conditions all the submitted jobs can complete.

REFERENCES

- [1] Zero configuration networking (zeroconf). WWW Page: <http://www.zeroconf.org>.
- [2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In Marios Mavronicolas and Philippos Tsigas, editors, *Proceedings of the 11th Workshop on Distributed Algorithms (WDAG'97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 1997.

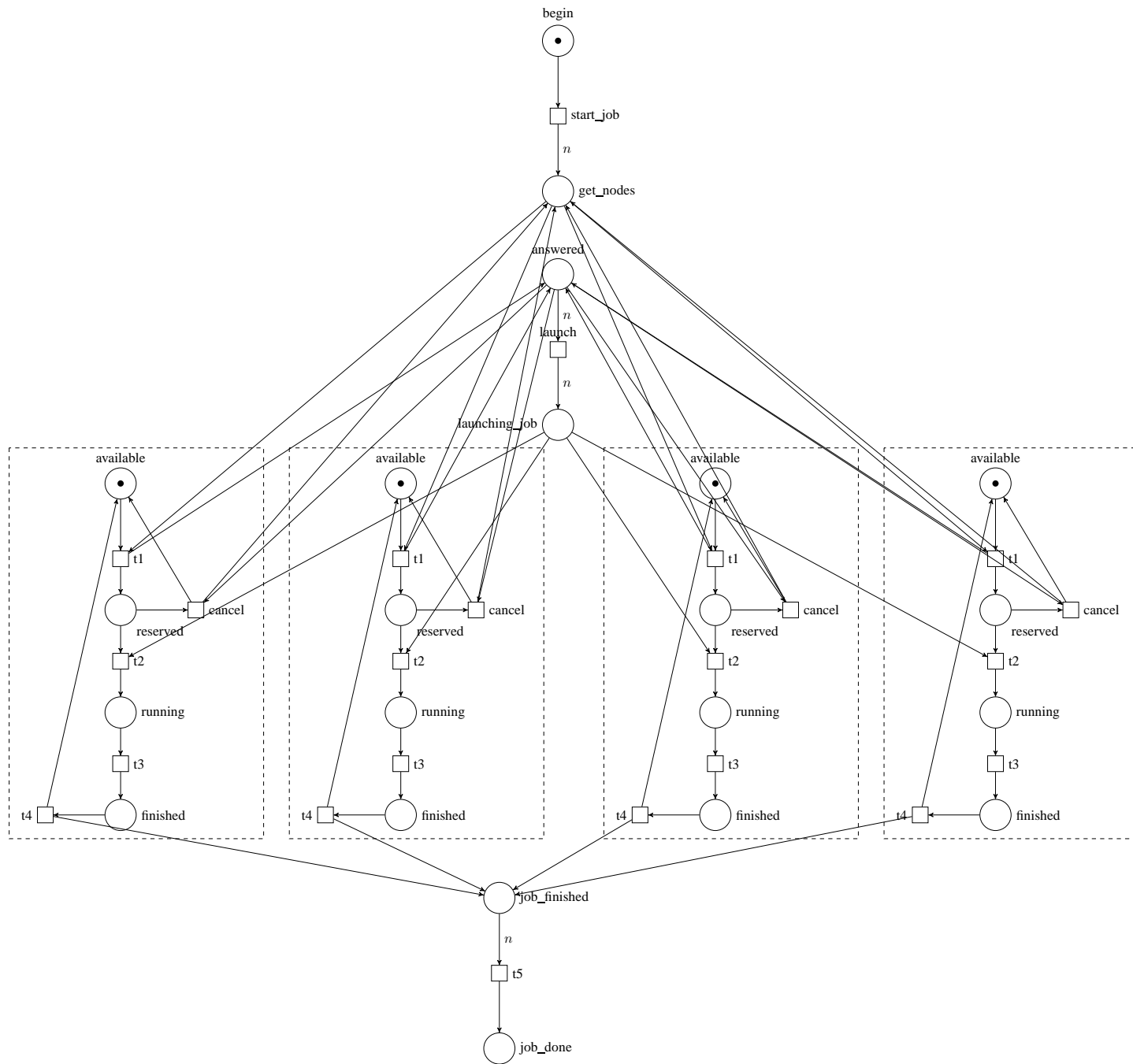


Fig. 8: Complete model for 4 machines and 1 job

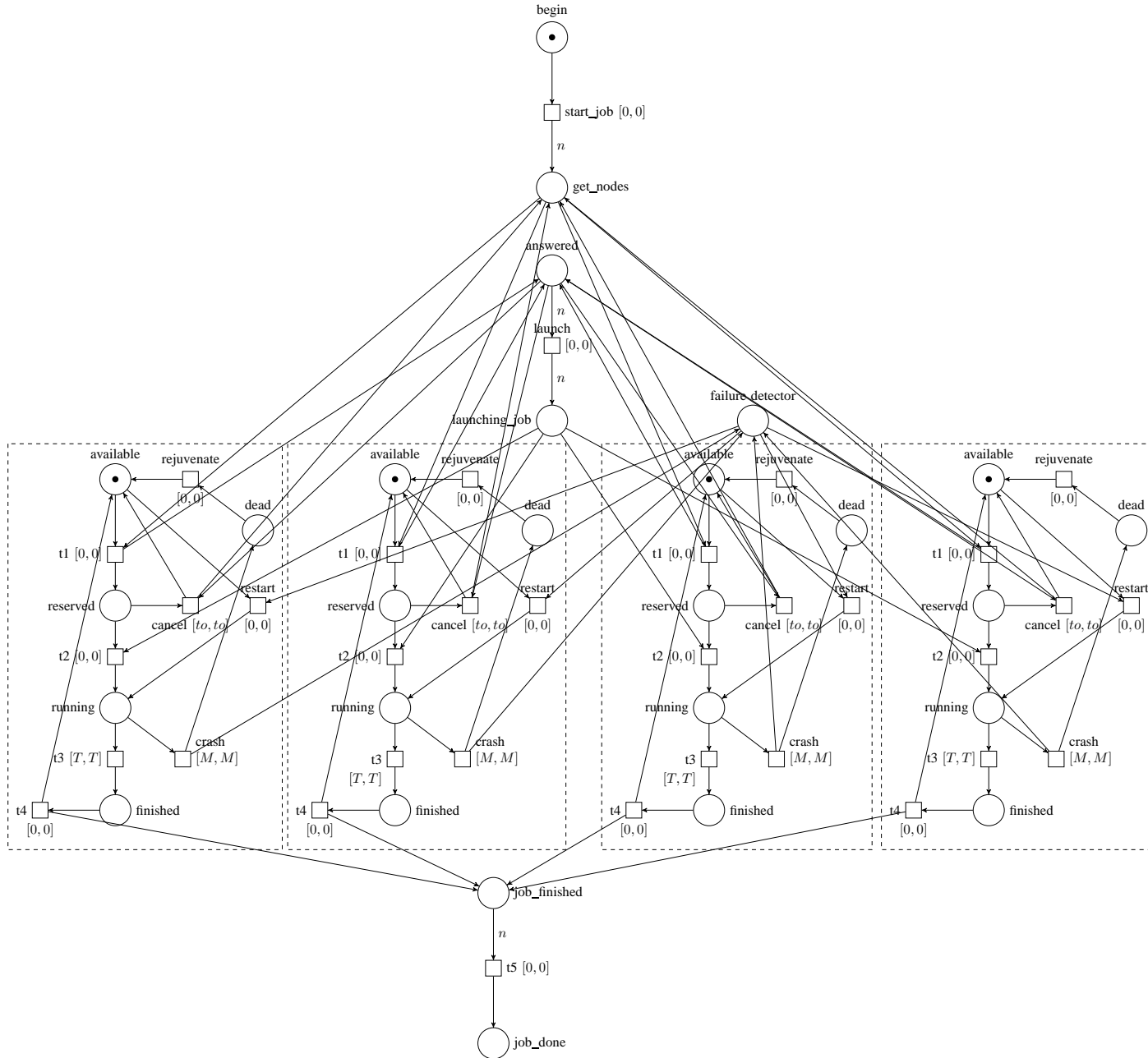


Fig. 9: Complete model for 4 machines and 1 job, with a failure detector

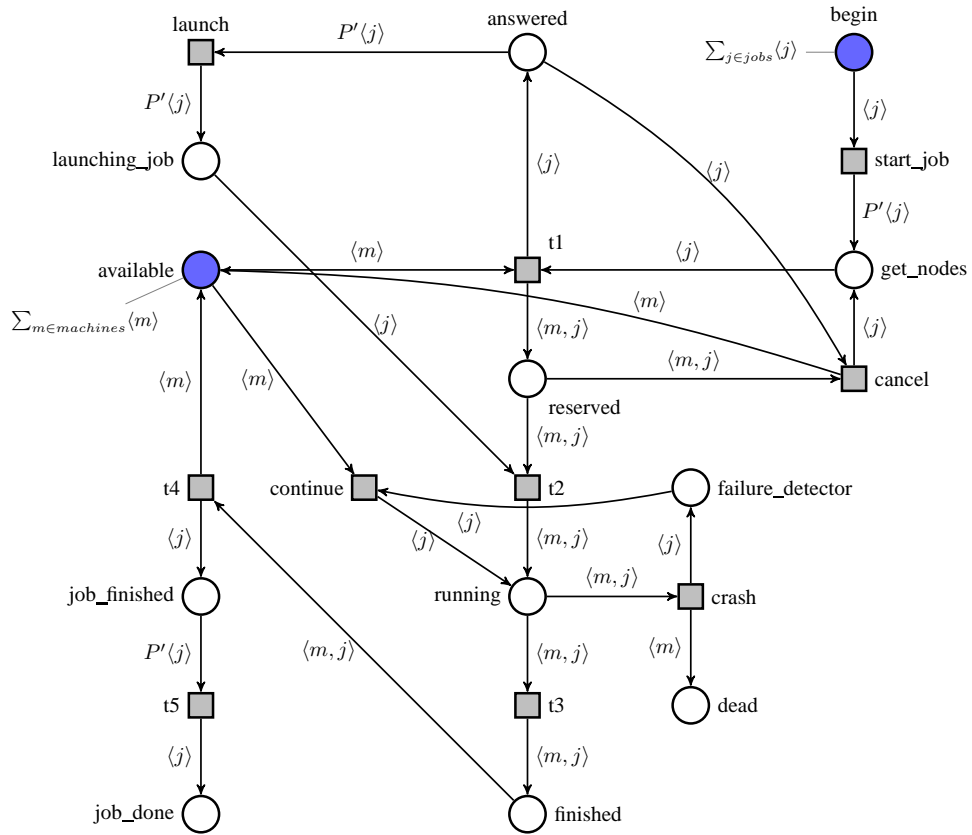


Fig. 10: Colored model